汉明码与 FSK 调制

无 02 江韩骏 2020010681
无 02 谢国强 2020010674
无 02 宋亦章 2020010675

分工情况：

江韩骏 信号发生以及并串/串并转换、代码整合

谢国强 FSK 调制与解调

宋亦章 (7,4)汉明码编码与解码

一、实验方案

实验使用汉明码和 FSK 调制进行数据的传输；

发送端由信号发生模块产生 8it 数据，进行汉明编码后增加帧头，进行并串转换，用 2 种不同频率的方波进行数字调制，接收端通过帧头完成同步接收、解调，之后进行解码，得到接收数据，对比误码率。其中发送和接收的 8bit 数据显示在 LED 上，调制完成之后的波形通过示波器查看。

系统框图可概括为：

8bit 信号产生->汉明码编码并增加帧头->并串转换->FSK 调制->信道->FSK 解调->串并转换->同步解码；

所有时钟由时钟分频模块得到，发送端和接收端各有一个时钟分频模块，即理论上可以做到异步收发；

二、关键代码与实验过程中遇到的问题及解决

模块及代码：

发送和接收都分别设计了顶层模块 Tx_top.v 和 Rx_top.v，并最终在 Top.v 顶层文件中例化；

1、时钟分频模块 clk_div.v

并串转换时钟模块需要是数据产生模块的 16 倍速才能保证数据的同步，同理调制模块的时钟也需要是并串转换模块的 16 倍速才能使用两种不同频率的方波完成 FSK 调制，因此需要时钟分频模块，基于系统时钟产生不同的时钟并分配；

2、信号发生模块 sig_gen.v

原先设计时想产生正弦信号，使用 matlab 生成了 256 个采样点生成定点数，调用 Vivado 的 Block Memory Generator IP 核例化 ROM，通过时钟访问地址完成正弦波数据的读出；实际上板的时候发现不知为何 IP 核总是报错，故验收的时候信号发生模块是顺次+1 产生数据；核心功能是每一定时间产生 8bit 数据，产生的数据具体是什么可以根据需要自行设计。

```
    input wire clk,
    input wire reset,
    output wire [7:0] sig_out
);
    reg [7:0] addr;
    always @(posedge clk) begin
      if(reset) begin
         addr <= 8'b0;
      end
      else begin
         addr <= addr + 8'b1;
      end
    end

    blk_mem_gen_0 sig_gnerator (
      .clka(clk),    // input wire clka
      .ena(1'b1),      // input wire ena
      .addra(addr),  // input wire [7 : 0] addra
      .douta(sig_out)  // output wire [7 : 0] douta
    );
```

3、汉明码编码/解码 hamming_encode.v & hamming_decode.v

采用(7,4)汉明码，对于输入的 8bit 数据，前 4bit 和后 4bit 分别进行编码，最后拼接为 14 位的编码后比特流，并在头部增加 2 位 1，作为帧头，故编码完成之后是 16bit；

解码部分对有效的 14bit 进行解码，得到 8bit 数据；

编码：编码结束之后发送之前在头 2 位添加’11’作为帧头；

```
`timescale 1ns / 1ps

module hamming_encode(
    input wire [7:0] bitstream,
    input wire clk,
    input wire rst,
    output wire [15:0] hammingcode
);
    reg [2:0] check1;
    reg [2:0] check2;
    reg [15:0] hammingcode_reg;
    assign hammingcode=hammingcode_reg;
always @(*) begin
    check1[2] <= bitstream[7] ^ bitstream[6] ^ bitstream[5];
    check1[1] <= bitstream[7] ^ bitstream[6] ^ bitstream[4];
    check1[0] <= bitstream[7] ^ bitstream[5] ^ bitstream[4];

    check2[2] <= bitstream[3] ^ bitstream[2] ^ bitstream[1];
    check2[1] <= bitstream[3] ^ bitstream[2] ^ bitstream[0];
    check2[0] <= bitstream[3] ^ bitstream[1] ^ bitstream[0];
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        hammingcode_reg <= 16'd0;
    end
    else hammingcode_reg <= {2'b11,bitstream[7:4], check1[2:0], bitstream[3:0], check2[2:0]};
end

endmodule
```

编码部分将输入的 8bit 看作 2 个 4bit，分别进行进行编码，最后再拼接发送；

解码：

```
always @(*) begin
    check1[2] <= code[13] ^ code[12] ^ code[11] ^ code[9];
    check1[1] <= code[13] ^ code[12] ^ code[10] ^ code[8];
    check1[0] <= code[13] ^ code[11] ^ code[10] ^ code[7];
    check2[2] <= code[6] ^ code[5] ^ code[4] ^ code[2];
    check2[1] <= code[6] ^ code[5] ^ code[3] ^ code[1];
    check2[0] <= code[6] ^ code[4] ^ code[3] ^ code[0];
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
      bitstream <= 4'd0;
    end
    else begin
      case(check1)
          3'b111: bitstream[7:4] <= {~code[13], code[12:10]};
          3'b110: bitstream[7:4] <= {code[13], ~code[12], code[11:10]};
          3'b101: bitstream[7:4] <= {code[13:12], ~code[11], code[10]};
          3'b011: bitstream[7:4] <= {code[13:11], ~code[10]};
          default: bitstream[7:4] <= code[13:10];
      endcase
      case(check2)
          3'b111: bitstream[3:0] <= {~code[6], code[5:3]};
          3'b110: bitstream[3:0] <= {code[6], ~code[5], code[4:3]};
          3'b101: bitstream[3:0] <= {code[6:5], ~code[4], code[3]};
          3'b011: bitstream[3:0] <= {code[6:4], ~code[3]};
          default: bitstream[3:0] <= code[6:3];
      endcase
    end
end

endmodule
```

4、并串/串并转换 P2S.v & S2P.v

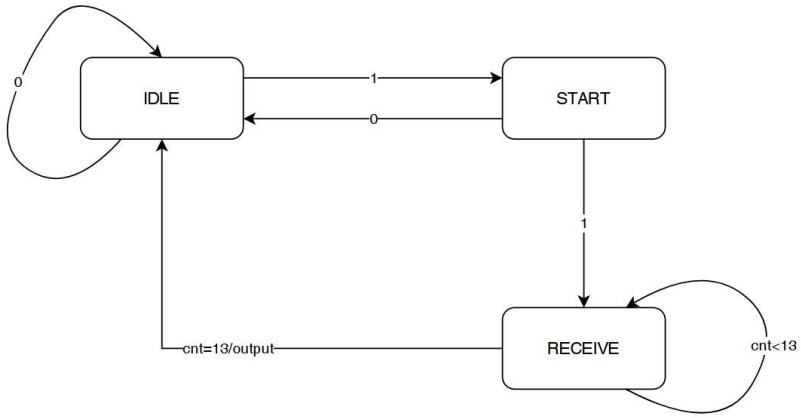
在编码完成之后，FSK 调制之前，需要将并行的 16bit 转为串行的比特串才能进行调制工作；

解调之后得到的比特串需要进行串并转换才能输入解码模块进行解码，在串并转换时通过状态机判断是否是一帧，接收完 1 帧之后，去除帧头，将 14bit 送入解码模块；

```
12
13      always@(posedge clk or posedge rst) begin
14        if(rst) begin
15          cnt <= 4'd0;
16          dout <= 1'b0;
17          sig_valid <= 1'b0;
18        end
19        else begin
20        if(din[15:14]==2'b11) begin
21          dout = din[15-cnt];
22          sig_valid<=1'b1;
23          if(cnt<=14) cnt <= cnt + 4'd1;
24          else cnt <= 4'd0;
25         end
26         else sig_valid<=1'b0;
27        end
28      end
29
30  endmodule
```

并串转换在检测到输入 16bit 数据的高 2 位为 11 的时候，分 16 个周期将这 1bit 串行输出；

串并转换状态机绘制如下：




```
always@(posedge clk or posedge rst) begin
  if(rst) begin
   current <= IDLE;
  end
  else begin
   current <= next;
  end
end

always@(*) begin
 case(current)
  IDLE:
    if(current_bit) next<=START;
    else next<=IDLE;
  START:
    if(current_bit) next<=RECEIVE;
    else next<=IDLE;//receive '11' is the beginning of a data
  RECEIVE:begin
    if(cnt<13) begin
     next <= RECEIVE;//receive 13bit
    end
    else begin
     //receive the last bit
     next <= IDLE;
    end
  end
  default:next <= IDLE;
 endcase
end
```

```
always@(posedge clk or posedge rst) begin
 if(rst) begin
  current_bit <= 1'b0;
  temp_reg <= 14'b0;
  cnt <= 16'd0;
  flag <= 1'b0;
 end
 else begin
  /*...*/
  current_bit<=din;
  case(current)
    IDLE:begin
     flag<=1'b0;
     temp_reg<=14'b0;
    end
    START:begin
     flag<=1'b0;
     temp_reg<=14'b0;
    end
    RECEIVE:begin
     temp_reg[13-cnt]<=current_bit;
     cnt<=cnt+16'd1;
     flag<=1'b0;
     if(cnt==13) begin
      flag<=1'b1;
      cnt<=16'd0;
     end
    end
    default:begin
     flag<=1'b0;
     temp_reg<=14'b0;
    end
  endcase
```

检测到 1 从 IDLE 进入 START 状态，如果又检测到 1，说明接收到了帧头，进入 RECEIVE 状态通过 cnt 进行计数，接收 14 个 bit 之后拉高 flag，将寄存器中的 14bit 输出；

5、FSK 调制/解调 fsk_modulate.v & fsk_demodulate.v

时钟分频产生 2 种不同频率的方波对比特流进行调制；

解调通过一个时钟周期内检测上升沿个数判断方波的频率，进而判断是 0 还是 1；

调制：

```
    reg [3:0] cnt;
    reg wave_h, wave_l;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            cnt <= 4'b0000;
            wave_h <= 0;
            wave_l <= 0;
        end
        else begin
         if(sig_valid) begin

            if(cnt == 4'b0111) begin
                wave_h <= ~wave_h;
                wave_l <= ~wave_l;

                cnt <= 4'b0000;
            end
            else if(~(cnt%4)) begin
                wave_h <= ~wave_h;
            end
            cnt = cnt + 1;
         end
         else cnt<=4'b0;
        end
    end

    assign wave_out = bit_in?wave_h:wave_l;

endmodule
```

调制部分通过两种频率的方波对比特流进行调制；

解调：

```
    always@(posedge clk or posedge rst)
    begin
        if (rst) begin
            last<=0;
            zeronum<=0;
            clknum<=5'b0;
        end
        else begin
            clknum<=clknum+1;
            if(in!=last)
            begin
                zeronum<=zeronum+1;
            end
            if(clknum == 3)
            begin
                if(zeronum > 1)
                begin
                    outreg<=1;
                end
                else
                begin
                    outreg<=0;
                end
            end
            if(clknum == 15)
            begin
                clknum<=0;
                zeronum<=0;
                out<=outreg;
            end
            last<=in;
        end
    end
```

解调部分通过检测边沿的个数判断对应的方波是高频还是低频；边沿检测方法为连续读入 2 位，看这 2 位是否相同，若不同则是边沿。采用的 2 种频率方波中，低频只会有一个边沿，因此边沿数量大于 1 则判定为高频，否则判定为低频。

实验问题与解决：

1、时钟的产生和同步

各个模块之间的时钟存在一定的倍率关系，假设我们每 1s 发送 1 个 8bit 数据，那么经过编码模块后相当于 1s 输出 16 个 bit，由于没有设置缓存，这 16bit 需要在 1s 内转为串行的比特流并发给调制，因此并串转换模块的时钟频率至少要是数据产生模块的 16 倍；同

时调制采用的是频率更高的方波，因此调制模块的时钟又要是并串转换模块的 16 倍，才能做到使用 2 倍和 8 倍并串转换时钟频率方波进行调制的效果。

我们组第一次验收就是因为时钟频率之间的倍数关系没有理清，导致数据不能及时传输，经过修改之后成功通过；

2、如何判断发送/接收的信号是否对齐

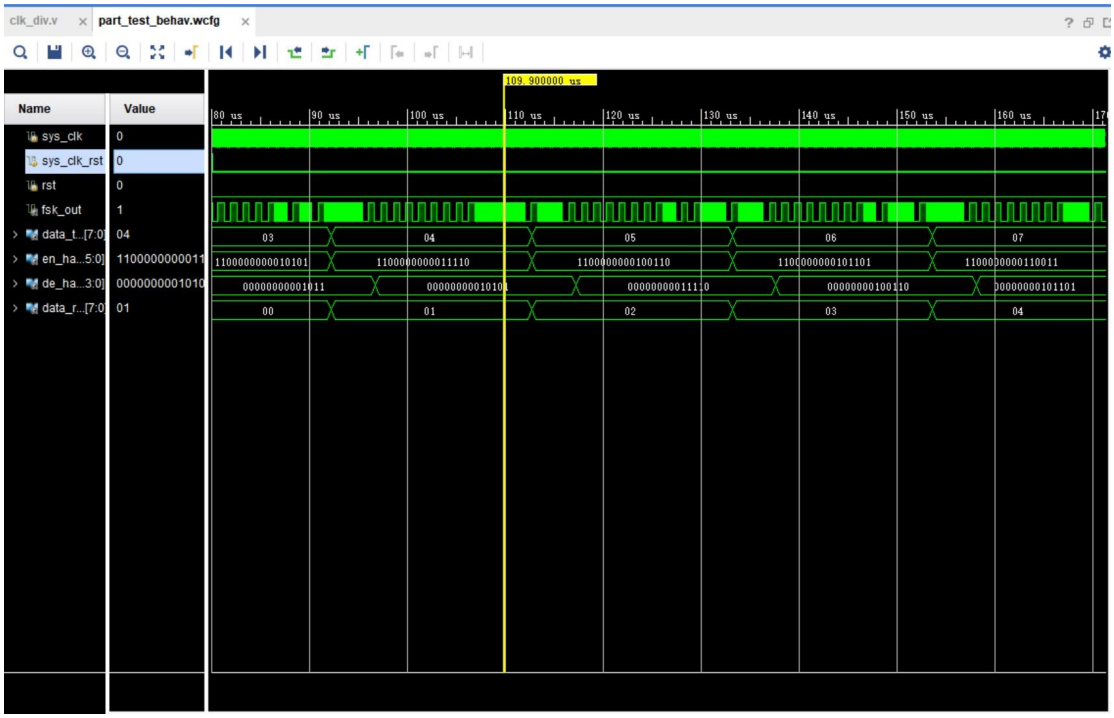
这个问题具体体现在并串/串并转换模块上。实际尝试的时候，我们一开始想利用握手机制完成，但发现其实只需要利用加入帧头的状态机就可以做到这一点。当然，现在的设计是不完美的，但在板内传输低误码率的情况下，几乎不会出现错判的情况。

3、复位

仿真中发现需要对时钟分频模块添加单独的复位信号，或者直接不添加复位信号。原因是其余所有模块的时钟信号均由时钟分频模块产生，若是所有复位均使用一个信号，那么复位之后由于时钟信号重新产生，将导致其余模块出现问题，因此时钟模块需要单独复位或者干脆不复位，避免影响其余模块的工作。

三、仿真结果

使用 Vivado 仿真如下所示：

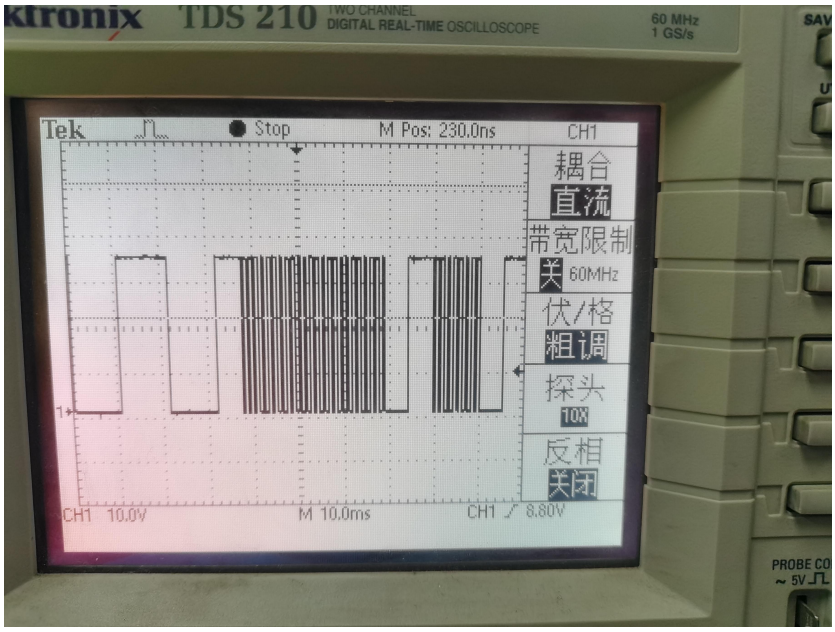


可以看到数据经过编码、调制之后完成了解调和解码，经过 2 个模块的延迟之后被输出；很明显能看出被 2 种不同频率方波调制的 fsk 信号，以及汉明编码前 2 位’11’的帧头；串并转化部分已经去掉了帧头，因此可以直接把 14bit 编码输入到解码器解码；

四、实验效果

实际上板的视频与报告、代码一起提交，其中 LED 灯是按照二进制依次变化的，发送和接收之间存在延迟；

其中调制之后的波形通过示波器显示如下：



能明显看出 2 种方波对 0 和 1 的调制；

五、总结

本次实验中最大的难点在于各个模块之间的接口规范和对接，这让我们感受到了在设计系统时并不像想象中那么容易，各个子系统之间的连接也很关键，需要一起合作才能最终搭建出一个完整的系统。我们也对相关的开发平台更加熟悉和了解，提升了自己的编程水平和硬件调试水平。

最后感谢老师和助教在这门课上给予我们的悉心指导！